Tutorials for queueing simulations

N.D. Van Foreest and E.R. Van Beesten

January 12, 2020

CONTENTS

1 TUTORIAL 2: SIMULATION OF THE $G/G/1$ QUEUE IN DISCRETE TIME

In this tutorial we simulate the queueing behavior of a supermarket or hospital, but it is easy to extend the ideas to much more general queueing system. We first make a simple model of a queueing system, and then extend this to cover more difficult queueing situations. With these models we can provide insight into how to design or improve real-world queueing systems. You will see, hopefully, how astonishingly easy it is to evaluate many types of decisions and design problems with simulation.

For ease we consider a queueing system in discrete time, and don't make a distinction between the number of jobs in the system and the number of jobs in queue. Thus, queue length corresponds here to all jobs in the system, cf. Section 1.4 of the queueing book.

1.1 *Set up*

1.1. Write down the recursions to compute the queue length at the end of a period based on the number of arrivals a_i during period i, the queue length Q_0 at the start, and the number of services s_i. Assume that service is provided at the start of the period. Then sketch an algorithm (in pseudo code) to carry out the computations (use a for loop). Then check this exercise's solution for the python code.

s.1.1. We need a few imports.

```python
import numpy as np
import scipy
from scipy.stats import poisson
import matplotlib.pyplot as plt

plt.ion()

scipy.random.seed(3)

def compute_Q_d(a, s, q0=0):
    d = np.zeros_like(a)
    Q = np.zeros_like(a)
    Q[0] = q0 # starting level of the queue
    for i in range(1, len(a)):
        d[i] = min(Q[i-1], s[i])
        Q[i] = Q[i-1] + a[i] - d[i]

    return Q, d
```

One of the nicest things about python is that the real code and pseudo code resemble each other so much.

With the code of the above exercises we can start our experiments.

1.2. Copy the code of the previous exercise to a new file in Anaconda. Then add the code below and run it. Here λ is the arrival rate, μ the service rate, N the number of periods, and q_0 the starting level of the queue. Explain what the code does. Can you also explain the value of the mean and the standard deviation?

```python
def experiment_1():
    labda, mu, q0, N = 5, 6, 0, 100
    a = poisson(labda).rvs(N)
    s = poisson(mu).rvs(N)
    print(a.mean(), a.std())

experiment_1()
```

1.3. Modify the appropriate parts of code of the previous exercise to the below, and run it. Explain what you see.

```python
def experiment_2():
    labda, mu, q0, N = 5, 6, 0, 100
    a = poisson(labda).rvs(N)
    s = poisson(mu).rvs(N)
    Q, d = compute_Q_d(a, s, q0)

    plt.plot(Q)
    plt.show()
    print(d.mean())


experiment_2()
```

s.1.3. You should see that the queue starts at 100 and drains roughly at rate $\lambda - \mu$. If you make Q_0 very large (10000 or so), you should see that the queue length process behaves nearly like a line until it hits 0.

1.4. Getting statistics is really easy now. For example, try to plot the empirical distribution function of the queue length process.

h.1.4. For the empirical distribution function you can use the code of Exercise **??**. Before looking it up, try to recall how the cdf is computed.

s.1.4. This is the code.

```python
def cdf(a):
    y = range(1, len(a) + 1)
    y = [yy / len(a) for yy in y]  # normalize
    x = sorted(a)
    return x, y

def experiment_3():
    labda, mu, q0, N = 5, 6, 0, 100
    a = poisson(labda).rvs(N)
    s = poisson(mu).rvs(N)
    Q, d = compute_Q_d(a, s, q0)

    print(d.mean())

    x, F = cdf(Q)
    plt.plot(x, F)
    plt.show()

experiment_3()
```

1.5. Can you explain the value of the mean number of departures?

s.1.5. The mean number of departures must be (about) equal to the mean number arrivals per period. Jobs cannot enter from ‘nowhere’.

1.6. Plot the queue length process for a large initial queue, for instance, with

```python
def experiment_4():
    labda, mu = 5, 6
    q0, N = 1000, 100
    a = poisson(labda).rvs(N)
    s = poisson(mu).rvs(N)
    Q, d = compute_Q_d(a, s, q0)
    plt.plot(Q)
    plt.show()


experiment_4()
```

Explain what you see.

s.1.6. The queue length drains at rate $\mu-\lambda$ when q_0 is really large. Thus, for such settings, you might just as well approximate the queue length behavior as $q(t)=q_0-(\mu-\lambda)t$, i.e., as a deterministic system.

1.7. Set $q_0=10000$ and $N=1000$. (In Anaconda you can just change the numbers and run the code again, in other words, you don't have to copy all the code.) Finally, make these values again 10 times larger.

Explain what you see. What is the drain rate of Q?

s.1.7. Copy this code and run it.

```
def experiment_5():
    N = 10000
    labda = 6
    mu = 5
    q0 = 0

    a = poisson(labda).rvs(N)
    s = poisson(mu).rvs(N)

    Q, d = compute_Q_d(a, s, q0)

    plt.plot(Q)
    plt.show()


experiment_5()
```

You should see that the queue drains with relatively less variations. The 'line' becomes 'straighter'.

1.8. What do you expect to see when $\lambda=6$ and $\mu=5$? Once you have formulated your hypothesis, check it.

s.1.8. Copy this code and run it.

```
def experiment_6():
    N = 10000
    labda = 5
    mu = 6
    q0 = 0

    a = poisson(labda).rvs(N)
    s = poisson(mu).rvs(N)

    Q, d = compute_Q_d(a, s, q0)
    print(Q.mean(), Q.std())

experiment_6()
```

The queue should increase with rate $\lambda - \mu$. Of course the queue length process will fluctuate a bit around the line with slope $\lambda - \mu$, but the line shows the trend.

1.2 *What-if analysis*

With simulation we can do all kinds of experiments to see whether the performance of a queueing system improves in the way we want. Here is a simple example of the type of experiments we can run now.

For instance, the mean and the sigma of the queue length might be too large, that is, customers complain about long waiting times. Suppose we are able, with significant technological investments, to make the service times more predictable. Then we would like to know the influence of this change on the queue length.

To quantify the effect of regularity of service times we first assume that the service times are exponentially distributed; then we change it to deterministic times. As deterministic service times are the best we can achieve, we cannot do any better than this by just making the service times more regular. If we are unhappy about its effect, we have to make service times shorter on average, or add extra servers, or block demand so that the inflow reduces.

1.9. Let's test the influence of service time variability. Replace the service times with the code `s = np.ones_like(a) * mu`. Read the docs of `numpy` to see what `np.ones_like` does. Explain the result.

s.1.9. Here is the code.

```
def experiment_6a():
    N = 10000
    labda = 5
    mu = 6
    q0 = 0

    a = poisson(labda).rvs(N)
    s = np.ones_like(a) * mu

    Q, d = compute_Q_d(a, s, q0)
    print(Q.mean(), Q.std())

experiment_6a()
```

You should observe that the mean and the variability of the queue length process decrease.

1.10. Next, we might be able to reduce the average service time by 10%, say, but reducing the variability is hard. To see the effect of this change, replace `s` by `s = poisson(1.1*mu).rvs(N)`, i.e., we increase the service rate with 10%.

Do the computations and compare the results to those of the previous exercise. What change in average service time do we need to get about the same average queue length as the one for the queueing system with deterministic service times? Is an 10% increase enough, or should it be 20%, or 30%? Just test a few numbers and see what you get.

s.1.10. Copy this code and run it.

```
def experiment_6b():
    N = 10000
    labda = 5
    mu = 6
    q0 = 0

    a = poisson(labda).rvs(N)
    s = poisson(1.1 * mu).rvs(N)

    Q, d = compute_Q_d(a, s, q0)
    print(Q.mean(), Q.std())

experiment_6b()
```

For our experiments it is about 20% extra.

You should make the crucial observation now that we can experiment with all kinds of changes (system improvements) and compare their effects. In more general terms: simulation allows us to do 'what-if' analyses.

1.3 *Control*

In the previous section we analyzed the effect of the design of the system, such as changing the average service times. These changes are independent of the queue length. In many systems, however, the service rate depends on the dynamics of the queue process. When the queue is large, service rates increase; when the queue is small, service rates decrease. This type of policy can often be seen in supermarkets: extra cashiers will open when the queue length increases.

Suppose that normally we have 6 servers, each working at rate 1 per period. When the queue becomes longer than 20 we hire two extra servers, and when the queue is empty again, we send the extra two servers home, until the queue hits 20 again, and so on.

1.11. Try to write pseudo-code (or a python program) to simulate the queue process. (This is pretty challenging; it's not a problem if you spend quite some time on this.)

h.1.11. As a hint, you need a state variable to track whether the extra servers are present or not. The solution shows the code.) Analyze the effect of the threshold at 20; what happens if you set it to 18, say, or 30? What is the effect of the number of extra servers; what if you would add 3 instead of 2?

s.1.11. Here is one way.

```python
def compute_Q_d_with_extra_servers(a, q0=0, mu=6, threshold=np.inf, extra=0):
    d = np.zeros_like(a)
    Q = np.zeros_like(a)
    Q[0] = q0
    present = False  # extra employees are not in
    for i in range(1, len(a)):
        rate = mu + extra if present else mu  # service rate
        s = poisson(rate).rvs()
        d[i] = min(Q[i - 1], s)
        Q[i] = Q[i - 1] + a[i] - d[i]
        if Q[i] == 0:
            present = False  # send employee home
        elif Q[i] >= threshold:
            present = True  # hire employee for next period

    return Q, d
```

```
def experiment_7():
    N = 10000
    labda = 5
    mu = 6
    q0 = 0

    a = poisson(labda).rvs(N)

    Q, d = compute_Q_d_with_extra_servers(a, q0, mu=6, threshold=20, extra=2)
    print(Q.mean(), Q.std())

    x, F = cdf(Q)
    plt.plot(x, F)
    plt.show()

experiment_7()
```

Note that this code runs significantly slower than the other code. This is because we now have to call `poisson(mu).rvs()` in every step of the for loop. We could make the program run faster by using the `scipy` library to generate N random numbers in one step outside of the for loop and then extract numbers from there when needed. However, for the sake of clarity we chose not to do so.

Another way to deal with large queues is to simply block customers if the queue is too long. What if we block at a level of 15? How would that affect the average queue and the distribution of the queue?

1.12. Modify the code to include blocking and do an experiment to see the effect on the queue length.

s.1.12. Here is an example. What can be the meaning of `np.inf`?

```
def compute_Q_d_blocking(a, s, q0=0, b=np.inf):
    # b is the blocking level.
    d = np.zeros_like(a)
    Q = np.zeros_like(a)
    Q[0] = q0
    for i in range(1, len(a)):
        d[i] = min(Q[i - 1], s[i])
        Q[i] = min(b, Q[i - 1] + a[i] - d[i])

    return Q, d



def experiment_7a():
    N = 10000
    labda = 5
    mu = 6
    q0 = 0

    a = poisson(labda).rvs(N)
    s = poisson(mu).rvs(N)

    Q, d = compute_Q_d_blocking(a, s, q0, b=15)
    print(Q.mean(), Q.std())
    x, F = cdf(Q)
    plt.plot(x, F)
    plt.show()


experiment_7a()
```

As a final case consider a single server queue that can be switched on and off. There is a cost h associated with keeping a job waiting for one period, there is a cost p to hire the server for one period, and it costs s to switch on the server. Given the parameter values of the next exercise, what would be a good threshold N such that the server is switched on when the queue hits or exceeds N? We assume (and it is easy to prove) that it is optimal to switch off the server when the queue is empty.

1.13. Jobs arrive at rate $\lambda = 0.3$ per period; if the server is present the service rate is $\mu = 1$ per period. The number of arrivals and service are Poisson distributed with the given rates. Then, $h = 1$ (without loss of generality), $p = 5$ and $S = 500$. Write a simulator to compute the average cost for setting $N = 100$. Then, change N to find a better value.

s.1.13. Here is all the code.

```
def compute_cost(a, mu, q0=0, threshold=np.inf, h=0, p=0, S=0):
    d = np.zeros_like(a)
    Q = np.zeros_like(a)
    Q[0] = q0
    present = False  # extra employee is not in.
    queueing_cost = 0
    server_cost = 0
    setup_cost = 0
    for i in range(1, len(a)):
        if present:
            server_cost += p
            c = poisson(mu).rvs()
        else:
            c = 0  # server not present, hence no service
        d[i] = min(Q[i - 1], c)
        Q[i] = Q[i - 1] + a[i] - d[i]
        if Q[i] == 0:
            present = False  # send employee home
        elif Q[i] >= threshold:
            present = True  # switch on server
            setup_cost += S
        queueing_cost += h * Q[i]

    print(queueing_cost, setup_cost, server_cost)

    total_cost = queueing_cost + server_cost + setup_cost
    num_periods = len(a) - 1
    average_cost = total_cost / num_periods
    return average_cost


def experiment_8():
    num_jobs = 10000
    labda = 0.3
    mu = 1
    q0 = 0
    threshold = 100  # threshold

    h = 1
    p = 5
    S = 500

    a = poisson(labda).rvs(num_jobs)
    av = compute_cost(a, mu, q0, threshold, h, p, S)
    print(av)


experiment_8()
```

After a bit of experimentation, we see that $N = 15$ is quite a bit better than $N = 100$.

1.14. What have you learned from this tutorial? What interesting extensions, relevant for practice, can you think of?

s.1.14. Some important points are the following.

1. Making a simulation requires some ingenuity, but is often not difficult.
2. With simulation it becomes possible to analyze many difficult queueing situations. The mathematical analysis is often much harder, if possible at all.
3. We studied the behavior of queues under certain control policies, typically policies that change the service rate as a function of the queue length. Such policies are used in practice, such as in supermarkets, hospitals, the customs services at airports, and so on. Hence, you now have some tools to design and analyze such systems.

An interesting extension is to incorporate time-varying demand. In many systems, such as supermarkets, the demand is not constant over the day. In such cases the planning of servers should take this into account.

1.4 *To be merged*

I copied this part from the book. To be merged. For students: you can skip this part.

1.15. Implement Eqs. (**??**) in Python and simulate a simple single-server queueing system. Assume that the $a_k \sim P(20)$, i.e., Poisson distributed with $\lambda = 20$, and $c_k = \mu = 21$. Make a plot of the queue length process Q, and compute the mean and standard variance of Q. Compute the fraction of periods in which the queue length exceeds some threshold, 20, say.

s.1.15. Below I fix the seed of the random number generator to ensure that I always get the same results from the simulator. The arrival process is Poisson, and the number of services is fixed to 21 per period. I use `Q = np.zeros_like(a)` to make an array of the same size as the number of arrival a initially set to zeros. The rest of the code is nearly identical to the formulas in the text.

```
>>> import numpy as np

>>> np.random.seed(3) # fix the seed

>>> labda = 20
>>> mu = 21
```

These are the number of arrivals in each period.

```
>>> a = np.random.poisson(labda, 10)
>>> a
array([21, 17, 14, 10, 22, 22, 17, 17, 19, 21])
```

The number of potential services

```
>>> c = mu * np.ones_like(a)
>>> c
array([21, 21, 21, 21, 21, 21, 21, 21, 21, 21])
```

Now for the queueing recursions:

```
>>> Q = np.zeros_like(a)
>>> d = np.zeros_like(a)
>>> Q[0] = 10  # initial queue length

>>> for k in range(1, len(a)):
...     d[k] = min(Q[k - 1], c[k])
...     Q[k] = Q[k - 1] - d[k] + a[k]
...
```

Here are the departures and queue lengths for each period:

```
>>> d
array([ 0, 10, 17, 14, 10, 21, 21, 19, 17, 19])
>>> Q
array([10, 17, 14, 10, 22, 23, 19, 17, 19, 21])
```

Suppose we define loss as the number of periods in which the queue length exceeds 20. Of course, any other threshold can be taken. Counting the number of such periods is very easy in Python: `(Q>20)` gives all entries of Q such $Q > 20$, the function `sum()` just adds them.

```
>>> loss = (Q > 20)
>>> loss
array([False, False, False, False,  True,  True, False, False, False,
        True])
>>> loss.sum()
3
```

Now all statistics:

```
>>> d.mean()
14.8
>>> Q.mean()
17.2
>>> Q.std()
4.377213725647858
>>> (Q > 20).sum()
3
```

Since this is a small example, the mean number of departures, i.e., `d.mean()`, is not equal to the arrival rate λ. Likewise for the computation of the mean, variance, and other statistical functions.

Now I am going to run the same code, but for a larger instance.

```
>>> num = 1000
>>> a = np.random.poisson(labda, num)
>>> c = mu * np.ones_like(a)
>>> Q = np.zeros_like(a)
>>> d = np.zeros_like(a)
>>> Q[0] = 10  # initial queue length

>>> for k in range(1, len(a)):
...     d[k] = min(Q[k - 1], c[k])
...     Q[k] = Q[k - 1] - d[k] + a[k]
...
>>> d.mean()
20.178
>>> Q.mean()
28.42
>>> Q.std()
10.155865300406461
>>> (Q > 30).sum()/num * 100
34.2
```

I multiply with 100 to get a percentage. Clearly, many jobs see a long queue, the mean is already some 28 jobs. For this arrival rate a service capacity of $\mu = 21$ is certainly too small.

Hopefully you understand from this discussion that once you have the recursions to construct/simulate the queueing process, you are 'in business'. The rest is easy: make plots, do some counting, i.e., assemble statistics, vary parameters for sensitivity analysis, and so on.

1.16. Change the service process of the previous exercise from $c_k = \mu = 21$ to $c_k \sim P(\mu)$ with $\mu = 21$. What is the influence of the higher variability of the service process on the queue length?

s.1.16. In this example the number of jobs served per day is $\sim P(21)$, so the service rate is still 21, but the period capacity c is a Poisson random variable.

```
>>> c = np.random.poisson(mu, num)
>>> Q = np.zeros_like(a)
>>> d = np.zeros_like(a)
>>> Q[0] = 10  # initial queue length

>>> for k in range(1, len(a)):
...     d[k] = min(Q[k - 1], c[k])
...     Q[k] = Q[k - 1] - d[k] + a[k]
...
>>> d.mean()
20.148
>>> Q.mean()
42.518
>>> Q.std()
22.841096208369684
>>> (Q > 30).sum()/num * 100
62.4
```

Comparing this to the result of the previous exercise in which the service capacity in each period was constant $c = 21$, we see that the average waiting time increases, just as the number of periods in which the queue length exceeds 30. Clearly, variability in service capacity does not improve the performance of the queueing system, quite on the contrary, it increases. In later sections we will see why this is so.